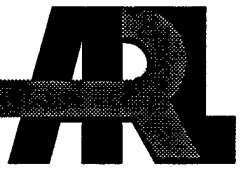
ARMY RESEARCH LABORATORY



MPI and HPF Performance in a DEC Alpha Cluster

by Dale Shires

ARL-TR-1668 May 1998

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5066

ARL-TR-1668 **May 1998**

MPI and HPF Performance in a DEC Alpha Cluster

Dale Shires
Weapons and Materials Research Directorate, ARL

Abstract

There are several different types of parallel computer architectures in use today. Some of these are large machines that house hundreds of processors with low- to mid-range computing capabilities. A different type of parallel computer architecture becoming increasingly popular is that of the cluster. Clusters are basically networked workstations: each containing 1 to ~30 processors with mid- to high-range computing capabilities. While arguments can be made for both paradigms, clusters seem to be gaining in popularity. They provide fast computation through multiplicity and fast processor throughput. Furthermore, code reuse on different cluster environments is now possible with the adoption of standard interprocessor communication tools like the message-passing interface (MPI) and high-performance FORTRAN (HPF). This report compares and contrasts the performance of MPI and HPF on a currently available computer cluster.

Contents

INTENTIONALLY LEFT BLANK.

List of Figures

INTENTIONALLY LEFT BLANK.

List of Tables

INTENTIONALLY LEFT BLANK.

Conventions.

`Typewriter font` is used for:

- any output directly produced by the computer (this includes program results and operating system messages), and
- program source code listings.

INTENTIONALLY LEFT BLANK.

1 Introduction.

The message-passing interface (MPI) and high-performance FORTRAN (HPF) are two sets of extensions to programming languages that seek to provide architecture-independent parallelism. MPI extensions exist for both the C and FORTRAN programming languages. HPF is specifically tailored to augment the FORTRAN 90 language. Both MPI and HPF have adopted published standards. HPF was adopted in 1993 [1], and MPI was standardized in 1994 [2]. These standards have allowed codes to be easily ported between different computer architectures with a minimum of code rewriting.

These two systems offer quite different parallel computational models. A computation in MPI usually consists of one or more processes that communicate messages (sends and receives) through calls to library routines. These libraries are ideally customized and optimized for the specific hardware system on which the computations are being performed. As long as the programs contain standard MPI directives, they should be able to be ported, recompiled, and executed on different architectures using the MPI system. MPI is sometimes referred to as the "assembly language" of parallel programming, since the programmer is required to specify the parallelism explicitly through calls to the message-passing library. This is often classified as an advantage rather than a hindrance. Effective cache use and memory management are becoming extremely important to achieving good parallel performance. Message passing can help in this regard by providing more programmer control of data locality in the memory hierarchy [3]. MPI appears to be surpassing the Parallel Virtual Machine (PVM) interface as the defacto standard in message passing parallelism.

In contrast, HPF is more closely associated with the data-parallel programming model. Data parallelism attempts to exploit the concurrency of the same operation to multiple data elements [4]. For example, one may wish to add the value 10 to each element of an array. This operation is inherently parallel, since there are no data dependencies inside this atomic operation. FORTRAN 90 supports such notations as A = A + 10 to perform this operation, where A can be a multidimensional array. HPF augments FORTRAN 90 with directives that inform the compiler that there are no data dependencies and the code region is data-parallel safe. It is up to the HPF compiler to optimally insert communication directives between processors and ensure synchronization between parallel regions. HPF is therefore slightly more abstract than MPI. It expresses parallelism at a relatively high level and is intended to remove the programmer from the more mundane tasks of specifying communication behavior between processors [5].

Of particular interest is how well these systems perform, both in general and compared to each other, in current computer architectures. The two basic performance metrics of parallel computing systems, speedup and efficiency, were studied in this experiment. Scalability is an interesting topic in this domain, but, as it relates to this problem, only provides useful data for algorithms with similar floating-point and

interprocessor communication requirements. The actual speedup achieved is defined as

$$S_p = \frac{T_l}{T_p}, \tag{1}$$

in which T_l is the linear (one processor) completion time for the algorithm, and T_p is the parallel completion time using p processors. The second metric used is efficiency. Efficiency is defined as

$$E_p = \frac{T_l}{p \cdot T_p}. \tag{2}$$

This number indicates the overall efficiency of the p processors working on the problem. Ideally, this number should be as close to 100% as possible, but will suffer because of conditions of load imbalancing, communication costs, and various other parallelization overheads.

The cluster architecture used in this experiment is quite relevant in today's computing environments. These coarse-grained architectures and clusters of workstations are becoming extremely popular for parallel computations. They can appear in many manifestations. Indeed, a cluster can be as little as two machines networked together. This alone, theoretically, boosts performance by a factor of 2. At the other end of the spectrum are very powerful hosts connected together over fast data channels, such as the Silicon Graphics Power Challenge Array (96 nodes) at the U.S. Army Research Laboratory. Three factors have played into clusters becoming viable parallel programming platforms. These factors are workstation-level high-performance microprocessors, standardized high-speed communication, and reliable standardized tools for distributed computing [6]. Today's workstations have processors that are quite robust. Very few of them need to be coupled to deliver impressive parallel performance. High-performance networks, such as ATM, HiPPI, and FDDI are now capable of delivering bandwidths of around 100 MB/s. Standard tools for synchronization and communication, such as MPI and HPF, continue to grow and mature into trusted systems.

2 Experimental Hypotheses.

Computer algorithms usually work on sets of data structures. Attacking these structures with multiple processors is what provides parallelism. Theoretically, if the data structures used are of size n, then n processors could be used to perform atomic operations in parallel. Breaking the problem down to find this lowest level of parallelism is known as decomposition. However, seldom do such large numbers of processors exist. For example, say we have an integer array B that is two-dimensional (2-D) and of size 1,000 $\times$ 1,000. While an operation B = B + 1 is possible in parallel, rarely do machines have one million processors. Therefore, the data must be agglomerated and distributed among the number of processors available. Several strategies and mechanisms are available for this task. HPF provides some standard decompositions,

while MPI requires the programmer to instrument the code to achieve the desired result. While a discussion of the numerous decompositions is not the purpose of this paper,* a quick example is helpful. Two agglomerations are quite popular in coarse-grained architectures where a minimum of communications is desired. These are one-dimensional (1-D) row and 1-D column decompositions and are are pictured in Figure 1. This figure shows the case of a 2-D array on a four-processor computer.

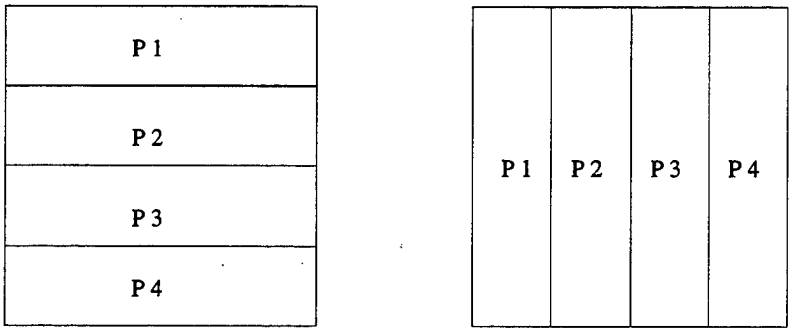


Figure 1: A 2-D Array Decomposed by Rows and Columns.

The 2-D array is blocked by rows in the row decomposition and by columns in the column decomposition. Processor 1 (P1) gets the first block of data, processor 2 (P2) the next, and so on. Data distribution is usually done to try and limit costly interprocessor communication. Other decompositions are available, but these two are quite prevalent in coarse-grained architectures. All of this depends, however, on the algorithm's action inside the data matrix.

Several decompositions and agglomerations were studied in this experiment. MPI requires data to be distributed to the individual processors explicitly by the programmer. HPF contains directives that specify how the data should be distributed by the compiler. These computations and tests were performed on a Digital Equipment Corporation (DEC) Alpha network with individual machines being connected by an FDDI network. The following hypotheses were investigated.

(1) One-dimensional row decomposition using MPI and 1-D column decomposition using HPF will result in the fastest speedups for this algorithm in the two different parallel programming environments. Row decomposition in C has shown to be slightly faster in studies of finite difference algorithms. This seems to be because the memory is laid out in row major order. In FORTRAN, one can expect column major order accessed by columns to be faster.

(2) There should be little difference between the fastest decomposition and agglomeration with MPI and the fastest similar decomposition with HPF. The main

*Complete examples can be found in [4] and [5].

limiting factor of both should be the speed of the interconnection network. Ideally, both should make the same optimal use of this system. FORTRAN code has shown itself to be faster in some algorithms, but the difference in the parallel codes using MPI and HPF should be no more than that noticed in the sequential algorithms.

(3) Based on studies of finite difference codes and the author's prior experience with other distributed memory, message-passing environments, an overall efficiency of about 85% is all that can be expected from the MPI and HPF codes.

3 Case Study: Image Matching.

3.1 The Image-Matching Algorithm.

The eye-brain system achieves three-dimensional (3-D) depth perception by taking advantage of two separate and distinct images captured by each eye. The image captured by the left eye is slightly different than the one captured by the right eye. This difference is called retinal disparity, and the brain is able to quickly use this information and other binocular cues to compute depth of objects in what the eyes are seeing.

Computers and machines can also determine depth of objects in their environments, but not quite as easily. One method involves using active sensors, or lasers, to determine range information. This method is limited in that lasers can usually only be directed at certain distinct points, thus limiting the machine's ability at determining range information in the entire scene.

A more interesting approach is to have the computer mimic the brain's behavior. This approach is known as computational stereo vision. A stereo camera pair is used to take pictures of a scene where range information is required. Because of binocular parallax, these cameras will acquire slightly different images of the scene, since they are at different locations. Points very distant from the cameras will appear to be at almost the same vertical and horizontal positions on digitized images from the cameras. Objects closer to the cameras will be more displaced. This phenomenon can be seen in Figure 2. Notice that points distant in the picture are at roughly the same location in both left and right images. However, points close to the camera, such as the measured white spot on the highway, have a much greater disparity in the two images.

By having the computer determine which points match in the two images, and then computing the amount the points shift, the computer is mimicing the retinal disparity computations performed by the brain. The mathematics involve only trigonometry; however, the number of transformations for high-resolution images becomes staggering. Every point is matched, thus giving a fine grid of range information that dwarfs the capabilities of active sensors. The main drawback of this approach, however, is

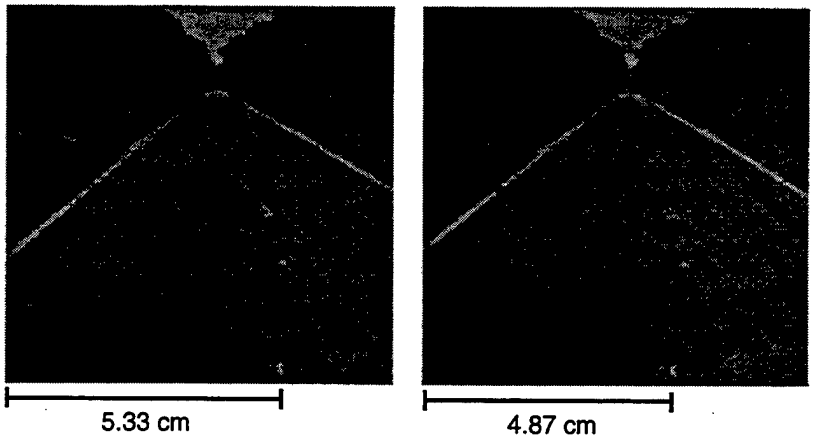


Figure 2: Depth Determined by Disparity in Images (ruler bars are not to scale).

the time it takes to match the points in the two images.

The algorithm to perform computational stereo is stochastic; therefore, it does require some time to complete. It is an undirected Monte Carlo search through the image space that produces a very fine, globally optimized disparity map where every pixel in one image is matched with its corresponding pixel in the stereo pair. As with other Monte Carlo algorithms, this approach requires a significant number of floating-point operations. However, the process of matching pixels typically requires only local interactions. On the computer, this translates into local references to memory by applying a nine-point stencil to the 2-D digitized images. Furthermore, besides certain boundary conditions, the amount of processing for each pixel remains uniform. These properties make the algorithm ideal for parallel processing.

Stereo matching requires global optimization. Since the digital image data maps pixel intensities to a relatively low resolution (typically 8 bits, implying 256 discrete levels), there are many possible matches in the local sense. Swatches in one image may appear to map other portions of the stereo image pair. To perform this operation accurately, the entire image must be taken into account.

One of the most popular optimization techniques to locate a global optimum is called simulated annealing. This approach can be applied directly to the image-matching problem [7]. As the name implies, the approach imitates a natural process. Annealing involves heating a solid to the extent that the molecules may randomly rearrange themselves and then cool gradually. Slowly lowering the temperature allows the molecules to settle into the lowest energy state, commonly described as thermal equilibrium. If the temperature rate declines too fast, defects may become frozen into the end state. If thermal equilibrium is maintained throughout the cooling cycle, the final system should be a globally optimized structure. For example, perfect crystals are grown in this manner.

The simulated annealing technique is outlined in Figure 3. The system is taken to

```
read I_R, I_L
D(row, col) = random number in [0 ... D_MAX]
T = T_MAX
/* loop according to fixed annealing schedule */
while T ≥ T_MIN
      S' ⇐ random state change S
      ΔE = E(S') − E(S)
      /* accept lower energy states */
      if ΔE < 0 then S = S'
      else
          P = e^(−ΔE/T)
          j = random number in [0 ... 1]
          /* accept higher energy only with Boltzman probability */
          if j < P then S = S'
      reduce T by predefined percentage
end while
```

Figure 3: Simulated Annealing Algorithm in Pseudocode.

equilibrium by the Metropolis algorithm by considering random, local state transitions on the basis of the change in energy that they imply. Since the system is stochastic, these local state changes can take the system away from convergence as well as toward it. This helps to prevent the system from sinking into local minima. The processing is complete when the system is in equilibrium at the lowest energy state achievable. A more detailed discussion of this technique and its Army applications may be found in [8].

The final result of the algorithm is a 2-D disparity map. The values in the disparity map are integer values ranging from 0 (no disparity) to D_MAX (maximum disparity). To better interpret the map, these values are coded with gray scale values and written to binary data files. Examples of these encoded disparity maps appear later in this paper.

3.2 Testing MPI and HPF for Correctness.

The stereo-matching algorithm was tested with several computer-generated random dot stereograms. These stereograms represent synthetic 3-D objects. In this case we simulate a camera system looking down on a "wedding cake" structure. Figure 4 shows the 3-D representation of this four-tiered structure. The stereogram is created by starting with a solid black background. It is then speckled with randomly placed white pixels. The number of white pixels is limited to 10% of the total image to test the robustness of the algorithm. The hypothetical right camera of the two-camera system is assigned this random dot image. Since the object is tiered, a stereo camera system above and facing the object would consist of two cameras (a right and left camera) that would perceive the dots to be at different locations in the two cameras.

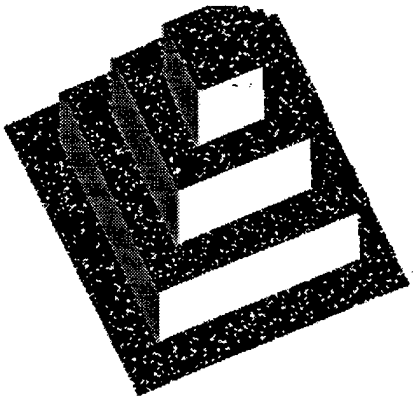


Figure 4: 3-D Wedding Cake Structure.

This effect is simulated by creating the left image of the stereo pair by shifting pixels in the right image to the right. Pixels around the outer edge were not offset, the next level in was offset by two pixels, the next level four pixels, and the center was offset by six pixels. Pixels with high movement represent areas that would be close to a camera looking down from above the wedding cake whereas pixels with no disparity would be distant from the camera. Figure 5 shows the random dot stereo pair.

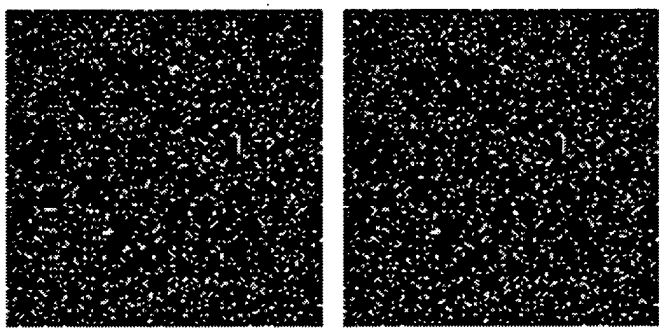


Figure 5: Random Dot Stereogram. The Left Image Is Formed by Displacing Points in the Right Image.

Because of the well-defined disparity maps, these random dot images represent ideal cases for evaluating stereo-matching algorithms. That is, we know how the result should look, whereas in a real-world image, there would be some doubt as to what an exact map should look like. There are still some areas of ambiguity, usually a result of sections that are devoid of white pixels; however, the overall structure of the map remains clear. The algorithm was checked for correctness and validated in all test cases. As an example, Figure 6 shows a gray scale encoded disparity map representing a solution to the matching problem from sequential C code and parallel

code using the MPI system. The actual 2-D result disparity map contains values in the range $0, \ldots, 6$. To better interprete the results, the final disparity maps were gray-scale encoded. Lighter areas in the image represent areas close to the camera; darker areas are further away.

4 Implementation Details.

4.1 MPI.

Implementation of the MPI version of the code was straightforward. The mpicc compiler on the DEC Alpha was used with linkage to the MPI libraries for access to the communication directives. The default optimization (-O) was used to correspond with the optimization used for the sequential code version. This ensures similar code generation and allows for meaningful comparisons between the sequential and parallel code versions.

One dimensional decompositions by rows and columns were used. The master process is in charge of opening and reading the left and right image data files. Since the disparity map is randomized at the start of the simulation, each processor initializes its own section of the disparity map. Image data never change during the program, so this static data is distributed once to each processor at the start of the program. Boundary conditions exist since a nine-point stencil is being passed over the 2-D arrays. For example, with the row decomposition shown in Figure 1, processor 1 requires the image data from the first row governed by processor 2. Also, processor 2 requires the last row of the image data owned by processor 1. Therefore, a processor also gets some of the image data that belongs to its neighboring processor at the beginning. However, since disparity data is dynamic, it must be distributed between processors at each loop iteration. A nine-point stencil is used in this algorithm because each point needs to know the disparity value of its eight neighboring points. This causes the creation of boundary conditions, or shadow points, along the processor boundaries. These shadow points define the interprocessor communication requirements for the computation.

Boundary conditions for the column decomposition are slightly more involved in terms of image data. Since the algorithm assumes a perfect image in the vertical orientation (no disparity up or down), we are only concerned with finding the horizontal disparity. The photometric component in the energy function uses the difference between the intensities of the proposed matched points in the left and right images. The point in the left image, however, can match a point in the right image up to D_MAX pixels to the right. For example, assume an image of size 256×256 with four processors working on the problem. Processor 1, with a column decomposition, will have image data from rows $0, \ldots, 255$ and columns $64, \ldots, 127$ of the left and right images. It could be possible that the point in the left image (row = 100, column = 127) matches with the point in the right image (100, 132). In this case, processor 1

does not have all of the data about the right image that it requires. To compensate for this in the column decompositions, each processor receives D_MAX extra columns of the right image from the master at the start of the algorithm.

The code was tested on random dot stereograms of sizes 200 × 200, 400 × 400, 600 × 600, and 800 × 800 with two, four, six, and eight processors. All combinations which were tested completed successfully. Figure 6 shows the 200 × 200 gray scale result maps for the sequential case, the MPI row decomposition, and the MPI column decomposition. The MPI results were generated in each case using four processors.

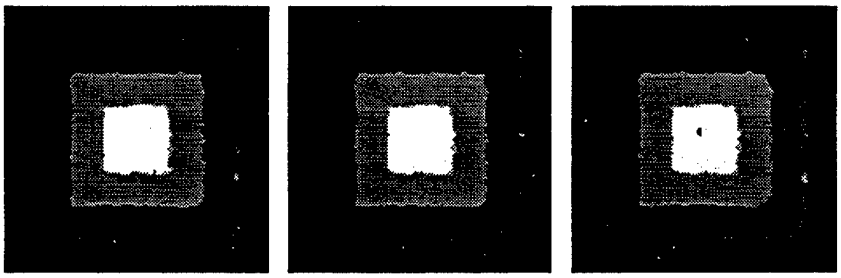


Figure 6: From Left to Right, Results From the Sequential C Program, MPI Row Decomposition, and MPI Column Decomposition. The Results Are Slightly Different for Each Case Given the Random Nature of the Simulations.

4.2 HPF.

Many difficulties were experienced while trying to move this code into FORTRAN 90 and into HPF. The HPF compiler on the DEC Alpha computers has not yet evolved to the stage of being a fully-functional HPF compiler. To begin with, several important FORTRAN 90 constructs, as well as critical HPF constructs are not supported. The compiler currently does not generate code tailored to the !HPF$ INDEPENDENT directive. Here is the compiler error message:

```
f90: Warning: rows.f90, line 148: The INDEPENDENT directive is checked for
syntactic and semantic correctness, but it is then ignored by the current
HPF compiler.
!HPF$ INDEPENDENT
```

This directive should be used around areas that are data-parallel safe (contains no dependencies) to assist the compiler in parallelizing the section. These directives were removed from the code to reduce the number of warning messages produced by the compiler. With a compiler that does support them, these directives should precede all of the FORALL statements in the simulated annealing code. Furthermore, WHERE statements may not be located inside of FORALL loops. This lack of functionality requires two FORALL constructs where often only one with a WHERE statement should be needed.

Data distribution directives in HPF are used to specify array data distribution to the processors available in the processor pool. As in MPI, two distributions were used, 1-D row and 1-D column. The first distribution used was !HPF$ DISTRIBUTE(BLOCK, *) :: dMap. This distributes the "dMap" array data in a row-blocked format, similar to the one used in MPI. The other distribution tested was (*, BLOCK) which corresponds to column decomposition. The rest of the bookkeeping and computational storage arrays (there were several) were aligned with the dMap array. There were several problems in getting HPF to work with this code. Several times, the executables would get hung up and then exit with:

```
TCP_MsgReadMsg: read error 54ows: ERROR Peer[0] (38)  _TCP_Send: send length
error - errno 9     (msgsend.c Line:377)
ows: ERROR Peer[2] (38)  _TCP_Send: send length error - errno 9 (msgsend.c
   Line:377)
ows: ERROR Peer[3] (38)  _TCP_Send: send length error - errno 9 (msgsend.c
   Line:377).
```

The (BLOCK, *) distribution produced the following error during runtime.

```
forrtl: error (72): floating overflow
TCP_MsgReadMsg: read error 54ows: ERROR Peer[0] (29)  _TCP_RecvAvail: Unexpected
EOF from peer 0         (msgrecva.c Line:189).
```

There appear to be no situations in the code that could cause such an error. Indeed, the type for the real array used in comparison to the exponential function was changed to double precision with the same results. Furthermore, the code performs fine with identical typing in the sequential version and in the HPF version in -single mode. Compiling the (*, BLOCK) distribution produced the following warnings:

```
f90: Warning: compute_loop_carried_sets: do_tree construct dt_group_forall
   not handled
f90: Warning: compute_loop_carried_sets: do_tree construct dt_group_forall
   not handled
f90: Warning: make_edges: do_tree construct dt_group_forall not handled
f90: Warning: make_edges: do_tree construct dt_group_forall not handled.
```

The initial belief was that having multiple statements inside a FORALL loop was the cause for these cryptic messages. However, the code does work properly when implemented on one processor. Also, HPF documentation states that multiple statements are allowed in FORALL loops and that they are performed in order. This should preclude any data dependencies.

Currently, there does not appear to be a good explanation for most of these warnings nor a good understanding of their intended meanings. Other distributions were attempted (e.g., CYCLIC), and each experienced some problem like those previously listed.

5 Experimental Results and Interpretation.

5.1 MPI.

Originally, the sequential code for the algorithm was written in C and compiled using the mpicc compiler. However, this produced executables that were slower than the MPI version with one processor. This resulted in dubious occurrences of superlinear speedup. A possible explanation resides in the compile sequence generated by the makefile that was used. The makefile performed the following operations:

```
mpicc -DFORTRANUNDERSCORE  -DMPE_USE_EXTENSIONS=1 -DHAS_XDR=1 -DSTDC_HEADERS=1
-DHAVE_STDLIB_H=1 -DMALLOC_RET_VOID=1 -DHAVE_SYSTEM=1 -DHAVE_NICE=1
-DPOINTER_64_BITS=1 -DINT_LT_POINTER=1 -DHAVE_LONG_DOUBLE=1
-DHAVE_LONG_LONG_INT=1 -O -DMPI_alpha -c mpirows.c
mpicc -O -o mpirows mpirows.o -L/usr/local/mpi-v1.0.12/lib/alpha/ch_p4  -lmpi
   -lm.
```

Undoubtedly, one of these options caused the production of more efficient code. Therefore, to make the process more homogeneous, the sequential version used in these comparisons is actually augmented with the MPI initializations and ran through MPI with one processor. Furthermore, to ensure a true reflection of speedup and efficiency, porsche, the faster processor of the group of DEC Alphas, was not used for these sequential timings. The sequential execution times for the different problem sizes are given in Table A.1. The code for this version of the algorithm is given in Appendix B. This is not a production code; therefore, comments and documentation within the code listings are sparse. The same caveat goes for all code listed in the appendices. The code for the row decomposition in MPI is given in Appendix C and the code for the column decomposition is given in Appendix D. The execution times for the row decompositions and column decompositions are given in Tables A.2 and A.3, respectively.

The speedup achieved by the MPI row decomposition is shown in Figure 7, and the speedup achieved by the MPI column decomposition is shown in Figure 8. To distribute the noncontiguous C data that results from column decompositions, a derived data type had to be created in MPI. MPI sends data based on addresses, which are row major in C. Trying to send data in column major order is undoubtedly simply adding more buffering behind the scenes. The speedups achieved by row decomposition are faster than the speedups for the column decomposition in every case.

The efficiencies corresponding to row and column decompositions are shown in Figures 9 and 10, respectively. The efficiencies are very good. The average row decomposition efficiency is around 94%. As more processors are invoked, one can see the efficiencies start to decline slightly. This is a standard phenomena. The code will only run as fast as the slowest processor in the pool. When using more processors, the individual load averages of the processors has a direct effect on efficiency. Still, when 94 out of every 100 instructions is working on the problem, the system has been well tuned.

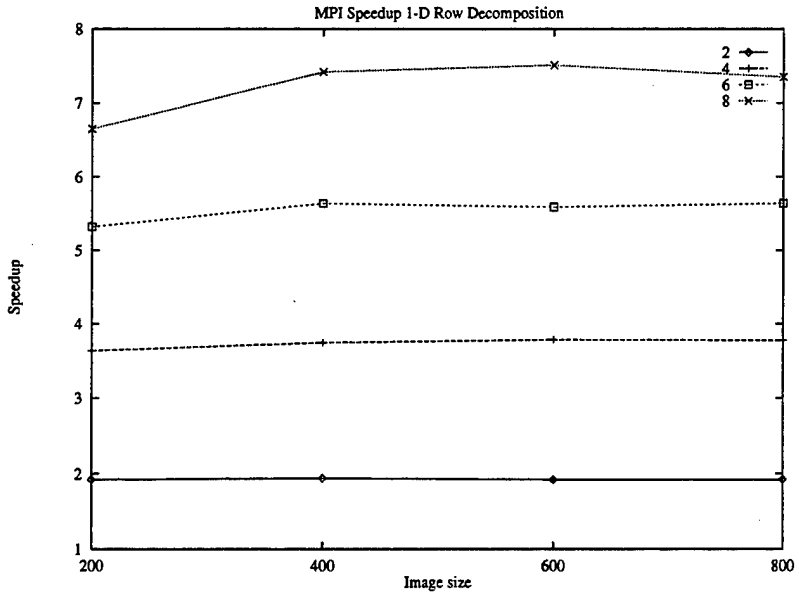


Figure 7: MPI 1-D Row Decomposition Speedup.

5.2 HPF.

The sequential timings were generated by compiling the HPF code using the standard f90 compiler. The HPF directives are ignored by the compiler, and sequential, one processor code is generated. There was a problem when trying to perform tests on the 800 × 800 image size. Dynamic array allocation was used for each array other than "dMap" since this array size must be known at compile time so the other arrays may be aligned to it. The reason for the runtime problems could not be determined, and this test case was omitted from the trials.

The code to perform the sequential, as well as the row and column decompositions in HPF, is given in Appendix E. The code listed performs row decomposition by distributing data in row block format. The column distribution was done by altering the HPF directive (BLOCK, *) to (*, BLOCK). Sequential code was generated by invoking the f90 compiler without HPF enabled. Table A.4 lists the sequential timing results.

Getting good results from HPF was almost impossible. Data distribution directives should affect only communication costs, not correctness. However, the directives did determine if the code would even execute or not. Each distribution performed properly when using one processor (executed with -single). Therefore, the following tables do list the times for one processor as well. Table A.5 lists the results for the row decomposition. Errors are noted in the table. Overflow means a floating-point overflow occurred. Exit means the program terminated prematurely with no warning or error message posted. EOF reflects a processor that received an unexpected end-of-file from another processor. Read errors indicate an error occurred while trying to

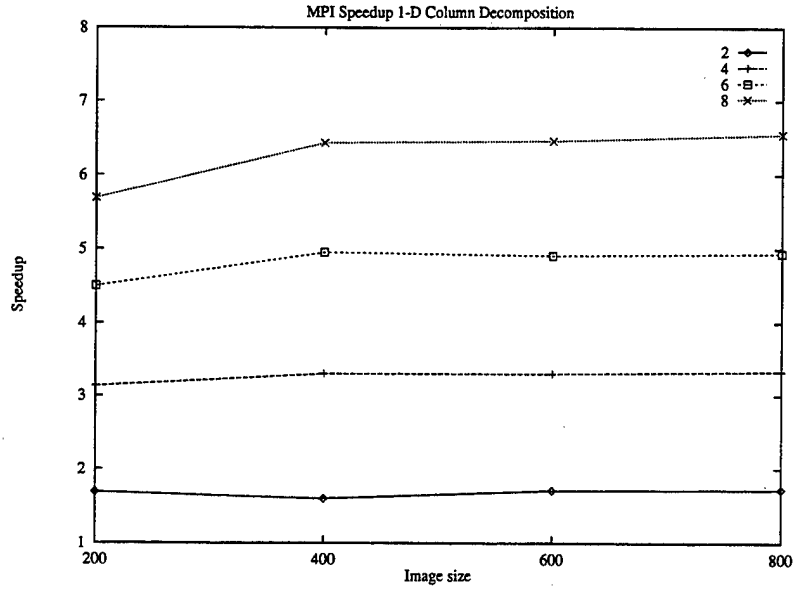


Figure 8: MPI 1-D Column Decomposition Speedup.

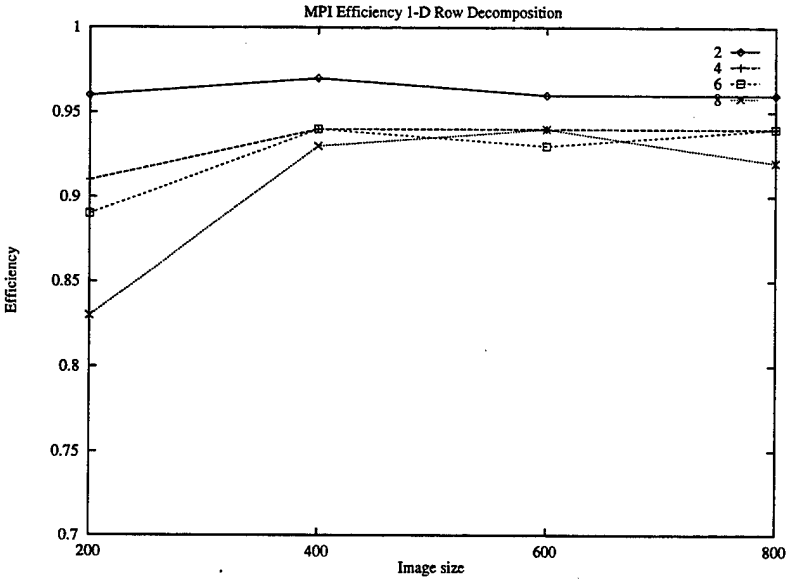


Figure 9: MPI 1-D Row Decomposition Efficiency.

read from a processor. Host indicates one of the hosts timed-out or died. Table A.6 lists the results for the column decomposition.

Why exactly there were so many problems with HPF remains a mystery. In cases where the code did perform, it did not perform well. There are several possible explanations. There could be too much overhead. This could be because of the way the single-program, multiple-data (SPMD) programs are created. Also, it might be

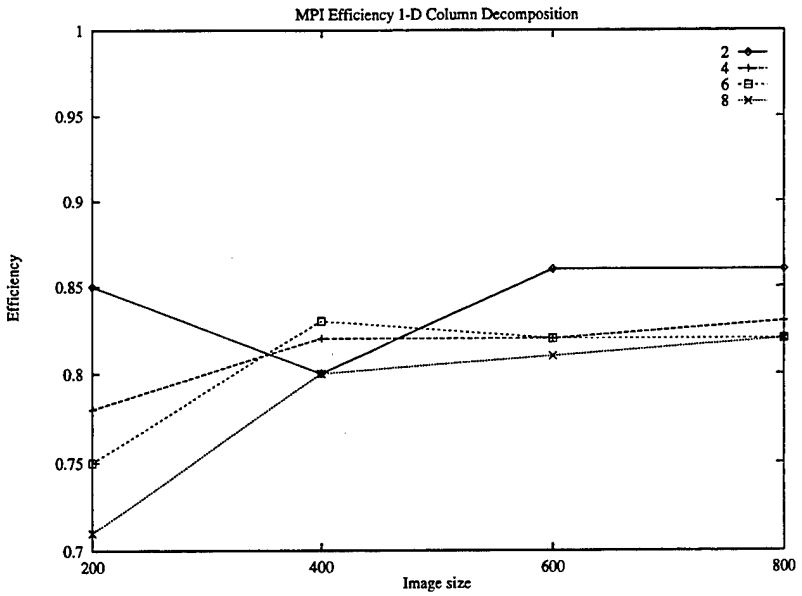


Figure 10: MPI 1-D Column Decomposition Efficiency.

that there is not enough work in the parallel regions. The cost of repeatedly calling the FORALL directive may be too high for the runtime system. Another possible reason is poor communication; possibly due to a poor communication library or unoptimized communication code. Unfortunately, in many cases it seemed that the compiler simply gave up and did not generate any communications upon reaching unsupported directives. The single processor code with parallel HPF directives activated was profiled. The results are given in Table A.7. While tracing down exact problems was difficult, it is evident from the overhead time that this was the major cause of degraded parallel performance.

6 Conclusion.

The experiments and testing did not entirely support the hypotheses. Indeed, 1-D row decomposition in MPI was faster than its column decomposition counterpart. This is probably because of the way MPI performs the derived data type. Copies to buffers to implement the strided column send will easily account for this difference. There is not enough good experimental data to comment on the HPF distributions. During implementation of the algorithm in HPF, it was realized that the hypothesis that column decomposition would be faster than row decomposition may be incorrect. Just as with MPI, the decomposition by columns does pose complications for HPF. When the data is decomposed by rows, each processor has all of the image data that is required of it. By columns, however, boundary conditions will have a greater impact, since the image data held by one processor might be required by its neighbor.

As far as there being little difference between the fastest MPI implementation

and the fastest HPF implementation, the results very clearly show that this is not the case. Comparisons between MPI and HPF prove almost impossible given the very poor performance of HPF. It is interesting to note, however, the much faster processing times achieved sequentially by FORTRAN 90 compared to C.

The efficiencies achieved by MPI were quite good. Prior experience with message-passing environments, such as C-Linda, and indeed some shared memory parallel systems such as the SGI, seemed to indicate a threshold of about 80–85% efficiency that could be expected of this type of code with multiple messages being passed at boundary conditions. While not tightly synchronized, the processors do have to operate in a lock-step type fashion, since updated boundary data must be computed and communicated at each iteration. Efficiencies around 94% in these cases indicate that MPI has been well thought out and implemented on this architecture.

The HPF system cannot currently compare to the MPI system for robustness and speed when operating on the DEC Alpha system. This seems to be the case, in general, as MPI is more quickly gaining acceptance as the message-passing system to use in cluster environments. MPI is a very good performer and should only get better as incremental changes and enhancements are made in the communication library. HPF will undoubtedly do better on stable, large array computations. This is usually the case with data parallel languages. HPF can improve dramatically by simply issuing better warnings and errors at both compile and run time. The HPF compiler will have to undergo major enhancements in its code generation section to be considered useful.

INTENTIONALLY LEFT BLANK.

References

[1] Center for Research on Parallel Computation, Houston, TX. *High Performance Fortran Language Specification, Version 1.0*, 1993.

[2] Computer Science Department, University of Tennessee, Knoxville, TN. *MPI: A Message-Passing Interface Standard*, 1994.

[3] GROPP, W., LUSK, E., AND SKJELLUM, A. *Using MPI: Portable Parallel Programming With the Message-Passing Interface*. The MIT Press, Cambridge, MA, 1994.

[4] FOSTER, I. *Designing and Building Parallel Programs*. Addison-Wesley Publishing Company, Sebastopol, CA, 1995.

[5] KOELBEL, C., LOVEMAN, D., AND SCHREIBER, R. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[6] PFISTER, G. F. *In Search of Clusters; The Coming Battle in Lowly Parallel Computing*. Prentice Hall, Upper Saddle River, NJ, 1995.

[7] BARNARD, S. T. *A stochastic approach to stereovision*. In *Readings in Computer Vision*. Addison-Wesley, New York, NY, 1987.

[8] SHIRES, D. R. *Exploiting parallelism in a monte carlo image-matching algorithm*. Technical Report ARL-TR-667, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, January 1995.

INTENTIONALLY LEFT BLANK.

A Tables

A.1 C Sequential Results. Times Are Listed in Seconds.

Image Size	Time
200 × 200	81.87
400 × 400	339.48
600 × 600	765.22
800 × 800	1365.54

A.2 MPI Row Decomposition Results. Times Are Listed in Seconds.

Image Size	Number Processors			
	2	4	6	8
200 × 200	42.67	22.56	15.38	12.31
400 × 400	175.44	90.65	60.24	45.77
600 × 600	397.79	202.45	136.77	101.83
800 × 800	711.12	361.91	242.29	185.77

A.3 MPI Column Decomposition Results. Times Are Listed in Seconds.

Image Size	Number Processors			
	2	4	6	8
200 × 200	48.39	26.18	18.19	14.38
400 × 400	212.68	102.88	68.55	52.77
600 × 600	446.54	232.01	155.99	118.43
800 × 800	792.14	409.80	276.43	208.44

A.4 FORTRAN Sequential Results. Times Are Listed in Seconds.

Image Size	Time
200 × 200	33.03
400 × 400	167.56
600 × 600	392.93

A.5 HPF (BLOCK, *) Distribution (Rows). Times Are Listed in Seconds.

Image Size	Number Processors				
	1	2	4	6	8
200 × 200	75.59	overflow	overflow	overflow	overflow
400 × 400	317.98	exit	EOF	overflow	read
600 × 600	734.99	exit	EOF	EOF	EOF

A.6 HPF (*, BLOCK) Distribution (Columns). Times Are Listed in Seconds.

Image Size	Number Processors				
	1	2	4	6	8
200 × 200	77.70	416.89	304.78	528.35	host
400 × 400	317.98	exit	exit	exit	host
600 × 600	734.99	exit	EOF	EOF	EOF

A.7 Single Processor HPF Profiling Results.

run-time statistics	peer 0 (value)	(value	minimum %skew	peer)	(value	maximum %skew	peer)
Timing Info. (sec)							
elapsed time	1008.23	1008.23	0.0	0	1008.23	0.0	0
profiling time	251.50	251.50	0.0	0	251.50	0.0	0
compute time	251.50	251.50	0.0	0	251.50	0.0	0
comm. time	0.00	0.00	0.0	0	0.00	0.0	0
active time	0.00	0.00	0.0	0	0.00	0.0	0
idle time	0.00	0.00	0.0	0	0.00	0.0	0
overhead time	779.34	779.34	0.0	0	779.34	0.0	0
user time	705.32	705.32	0.0	0	705.32	0.0	0
system time	298.34	298.34	0.0	0	298.34	0.0	0.

INTENTIONALLY LEFT BLANK.

B C sequential code.

```
#include <math.h>
#include <stdio.h>
#include <sys/time.h>
#include <mpi.h>

#define MAX_ROWS 1050
#define MAX_COLS 1050
#define D_MIN 0
#define D_MAX 6
#define LAMBDA 5
#define STARTING_TEMP 100.0
#define ENDING_TEMP 1.0
#define TEMP_REDUCTION 0.1
#define LATTICE_SCANS 10

/* Define various macros and functions to inline code. */

#define RANDOM_DISPARITY (rand() % D_MAX)
#define RANDOM_PROBABILITY (rand() % 32767 / 32767.0)
#define min(x,y) ((x) < (y)) ? (x) : (y)
#define max(x,y) ((x) > (y)) ? (x) : (y)

typedef unsigned char Pixel;
enum {LEFT = 0, RIGHT};

/* Global variables: */

Pixel leftImage[MAX_ROWS][MAX_COLS];
Pixel rightImage[MAX_ROWS][MAX_COLS];
int disparityMap[MAX_ROWS][MAX_COLS];
int tempDisparityMap[MAX_ROWS][MAX_COLS];

/* Function prototypes: */

/* Basic file I/O routines. */

static int ReadLeftImage(int n);
static int ReadRightImage(int n);
static void WriteResultGrayScaleMap(int n);

/* Simulated annealing functions. */

static int RandomNewState(int oldState);
static void Anneal(int n);

void main(int argc, char *argv[]);


/*****************************************************************************
   File I/O routines follow.
*****************************************************************************/

static int ReadLeftImage(int n) {
   FILE *theFile;
   char fileName[80];
   int i, j;

   /* Attempt to read the left image. */

   printf("Reading left image.\n");
```

```
   sprintf(fileName, "LeftDot%d.c", n);

   theFile = fopen(fileName, "r");

   if (theFile == NULL) {
      perror("ReadLeftImage: fopen");
      return(FALSE);
      }

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         fscanf(theFile, "%c", &(leftImage[i][j]));

   fclose(theFile);

   return(TRUE);

   } /* end ReadLeftImage */


static int ReadRightImage(int n) {
   FILE *theFile;
   char fileName[100];
   int i, j;

   /* Attempt to read the right image. */

   printf("Reading right image.\n");

   sprintf(fileName, "RightDot%d.c", n);

   theFile = fopen(fileName, "r");

   if (theFile == NULL) {
      perror("ReadRightImage: fopen");
      return(FALSE);
      }

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         fscanf(theFile, "%c", &(rightImage[i][j]));

   fclose(theFile);

   return(TRUE);

   } /* end ReadRightImage */


static void WriteResultGrayScaleMap(int n) {
   FILE *theFile;
   char fileName[80];
   int i, j;

   sprintf(fileName, "C_SEQ_RESULTS_%d", n);

   theFile = fopen(fileName, "w");

   if (theFile == NULL) {
      perror("WriteResult: fopen");
      exit(1);
      }

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
```

```
            fprintf(theFile, "%c", (Pixel)(disparityMap[i][j] * (255 / D_MAX)));

      if (fclose(theFile) != 0)
         perror("WriteResult: fclose");

      } /* end WriteResultGrayScaleMap */


/*****************************************************************************
   Stereo matching routines follow.
*****************************************************************************/

static int RandomNewState(int oldState)
   {
   int randomNumber;

   if (RANDOM_DISPARITY > ((D_MAX / 2) - 1))
      randomNumber = -1;
   else
      randomNumber = 1;

   oldState = oldState + randomNumber;

   if (oldState < D_MIN)
      oldState = D_MIN;
   else if (oldState > D_MAX)
      oldState = D_MAX;

   return(oldState);

   } /* end RandomNewState */


static void RandomizeSystem(int n) {
   int i, j;

   /* Assign a random disparity to each disparity grid point.  Edges get
      the value 0. */

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++) {
         if ((i == 0) || (i == (n - 1)) || (j == 0) || (j >= (n - D_MAX)))
            disparityMap[i][j] = 0;
         else
            disparityMap[i][j] = RANDOM_DISPARITY;
         }

   } /* end RandomizeSystem */


static int Energy(int row, int col, int disparity) {
   int delta;
   int photometric;

   delta = abs(disparity - disparityMap[row-1][col-1]) +
           abs(disparity - disparityMap[row-1][col]) +
           abs(disparity - disparityMap[row-1][col+1]) +
           abs(disparity - disparityMap[row][col-1]) +
           abs(disparity - disparityMap[row][col+1]) +
           abs(disparity - disparityMap[row+1][col-1]) +
           abs(disparity - disparityMap[row+1][col]) +
           abs(disparity - disparityMap[row+1][col+1]);

   photometric = abs(leftImage[row][col+disparity] - rightImage[row][col]);
```

```
   return(photometric + (LAMBDA * delta));

   } /* end Energy */


static void Anneal(int n) {
   int newState, newEnergy, oldEnergy, i, j;
   double currentTemp;
   int scanCounter;
   int deltaEnergy;

   RandomizeSystem(n);

   currentTemp = STARTING_TEMP;

   while (currentTemp >= ENDING_TEMP) {

      printf("Temp = %f\n", currentTemp);

      for (scanCounter = 0; scanCounter < LATTICE_SCANS; scanCounter++) {
         for (i = 1; i < (n - 1); i++)
            for (j = 1; j < (n - D_MAX); j++) {
               newState = RandomNewState(disparityMap[i][j]);
               oldEnergy = Energy(i, j, disparityMap[i][j]);
               newEnergy = Energy(i, j, newState);
               deltaEnergy = newEnergy - oldEnergy;
               if (deltaEnergy < 0)
                  tempDisparityMap[i][j] = newState;
               else if (RANDOM_PROBABILITY <
                  exp((double)-deltaEnergy/currentTemp))
                  tempDisparityMap[i][j] = newState;
               else
                  tempDisparityMap[i][j] = disparityMap[i][j];
               }

         for (i = 1; i < (n - 1); i++)
            for (j = 1; j < (n - D_MAX); j++)
               disparityMap[i][j] = tempDisparityMap[i][j];

         }
      currentTemp -= (currentTemp * TEMP_REDUCTION);
      }

   } /* end Anneal */


void main(int argc, char *argv[]) {
   int n, i;
   struct timespec t1, t2;
   double startTime, endTime;


   MPI_Init(&argc, &argv);
   n = atoi(argv[1]);

   if (ReadLeftImage(n) && ReadRightImage(n)) {
      getclock(TIMEOFDAY, &t1);
      startTime = (t1.tv_sec * 100.0) + (t1.tv_nsec * 10e-7);
      startTime = MPI_Wtime();
      Anneal(n);
      getclock(TIMEOFDAY, &t2);
      endTime = (t2.tv_sec * 100.0) + (t2.tv_nsec * 10e-7);
      printf("mpi %f\n", MPI_Wtime() - startTime);

      }
```

```
printf("Elapsed time = %.2f seconds\n", (endTime - startTime) / 100.0);
WriteResultGrayScaleMap(n);

MPI_Finalize();

} /* end main */
```

INTENTIONALLY LEFT BLANK.

C MPI 1-D row decomposition.

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <sys/time.h>

#define MAX_ROWS 1000
#define MAX_COLS 1000
#define D_MIN 0
#define D_MAX 6
#define LAMBDA 5
#define STARTING_TEMP 100.0
#define ENDING_TEMP 1.0
#define TEMP_REDUCTION 0.1
#define LATTICE_SCANS 10

/* Define various macros and functions to inline code. */

#define RANDOM_DISPARITY (rand() % D_MAX)
#define RANDOM_PROBABILITY (rand() % 32767 / 32767.0)
#define min(x,y) ((x) < (y)) ? (x) : (y)
#define max(x,y) ((x) > (y)) ? (x) : (y)

typedef unsigned char Pixel;
enum {LEFT = 0, RIGHT};

/* Global variables: */

Pixel leftImage[MAX_ROWS][MAX_COLS];
Pixel rightImage[MAX_ROWS][MAX_COLS];
int disparityMap[MAX_ROWS][MAX_COLS];
int tempDisparityMap[MAX_ROWS][MAX_COLS];

/* Function prototypes: */

/* Basic file I/O routines. */

static int ReadLeftImage(int n);
static int ReadRightImage(int n);
static void WriteResultGrayScaleMap(int n);

/* MPI communication routines. */

static void GetImageDataFromMaster(int n, int myID, int startRow, int stopRow);
static void SendResultsToMaster(int n, int myID, int startRow, int stopRow);
static void CollectResults(int n, int numProcs, int chunkSize);

/* Simulated annealing functions. */

static int RandomNewState(int oldState);
static void Anneal(int n, int myID, int startRow, int stopRow, int numProcs);

void main(int argc, char *argv[]);


/*****************************************************************************
   File I/O routines follow.
*****************************************************************************/

static int ReadLeftImage(int n) {
   FILE *theFile;
   char fileName[80];
```

```
   int i, j;

   /* Attempt to read the left image. */

   sprintf(fileName, "LeftDot%d.c", n);

   theFile = fopen(fileName, "r");

   if (theFile == NULL) {
      perror("ReadLeftImage: fopen");
      return(FALSE);
      }

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         fscanf(theFile, "%c", &(leftImage[i][j]));

   fclose(theFile);

   return(TRUE);

   } /* end ReadLeftImage */


static int ReadRightImage(int n) {
   FILE *theFile;
   char fileName[100];
   int i, j;

   /* Attempt to read the right image. */

   sprintf(fileName, "RightDot%d.c", n);

   theFile = fopen(fileName, "r");

   if (theFile == NULL) {
      perror("ReadRightImage: fopen");
      return(FALSE);
      }

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         fscanf(theFile, "%c", &(rightImage[i][j]));

   fclose(theFile);

   return(TRUE);

   } /* end ReadRightImage */


static void WriteResultGrayScaleMap(int n) {
   FILE *theFile;
   char fileName[80];
   int i, j;

   sprintf(fileName, "MPI_ROWS_RESULTS_%d", n);

   theFile = fopen(fileName, "w");

   if (theFile == NULL) {
      perror("WriteResult: fopen");
      exit(1);
      }
```

```
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         fprintf(theFile, "%c", (Pixel)(disparityMap[i][j] * (255 / D_MAX)));

   fclose(theFile);

   } /* end WriteResultGrayScaleMap */


/*****************************************************************************
   MPI communication routines follow.
*****************************************************************************/

static void CollectResults(int n, int numProcs, int chunkSize) {
   int i, source;
   MPI_Status status;

   for (i = chunkSize; i < n; i++) {
      source = min(i / chunkSize, numProcs - 1);
      MPI_Recv(disparityMap[i], n, MPI_INT, source, source, MPI_COMM_WORLD,
         &status);
      }

   } /* end CollectResults */


static void GetImageDataFromMaster(int n, int myID, int startRow, int stopRow) {
   MPI_Status status;
   int i;

   for (i = startRow; i <= stopRow; i++) {
      MPI_Recv(leftImage[i], n, MPI_UNSIGNED_CHAR, 0, LEFT, MPI_COMM_WORLD,
         &status);
      MPI_Recv(rightImage[i], n, MPI_UNSIGNED_CHAR, 0, RIGHT, MPI_COMM_WORLD,
         &status);
      }

   } /* end GetImageDataFromMaster */


static void SendResultsToMaster(int n, int myID, int startRow, int stopRow) {
   int i;

   for (i = startRow; i <= stopRow; i++)
      MPI_Send(disparityMap[i], n, MPI_INT, 0, myID, MPI_COMM_WORLD);

   } /* end SendResultsToMaster */


/*****************************************************************************
   Stereo matching routines follow.
*****************************************************************************/

static int RandomNewState(int oldState)
   {
   int randomNumber;

   if (RANDOM_DISPARITY > ((D_MAX / 2) - 1))
      randomNumber = -1;
   else
      randomNumber = 1;

   oldState = oldState + randomNumber;

   if (oldState < D_MIN)
```

```
      oldState = D_MIN;
   else if (oldState > D_MAX)
      oldState = D_MAX;

   return(oldState);

   } /* end RandomNewState */


static void RandomizeSystem(int startRow, int stopRow, int n) {
   int i, j;

   /* Assign a random disparity to each disparity grid point.  Edges get
      the value 0. */

   for (i = startRow; i <= stopRow; i++)
      for (j = 0; j < n; j++) {
         if ((i == 0) || (i == (n - 1)) || (j == 0) ||  (j >= (n - D_MAX)))
            disparityMap[i][j] = 0;
         else
            disparityMap[i][j] = RANDOM_DISPARITY;
         }

   } /* end RandomizeSystem */


static int Energy(int row, int col, int disparity) {
   int delta;
   int photometric;

   delta = abs(disparity - disparityMap[row-1][col-1]) +
           abs(disparity - disparityMap[row-1][col]) +
           abs(disparity - disparityMap[row-1][col+1]) +
           abs(disparity - disparityMap[row][col-1]) +
           abs(disparity - disparityMap[row][col+1]) +
           abs(disparity - disparityMap[row+1][col-1]) +
           abs(disparity - disparityMap[row+1][col]) +
           abs(disparity - disparityMap[row+1][col+1]);

   photometric = abs(leftImage[row][col+disparity] - rightImage[row][col]);

   return(photometric + (LAMBDA * delta));

   } /* end Energy */


static void Anneal(int n, int myID, int startRow, int stopRow, int numProcs) {
   int newState, newEnergy, oldEnergy, i, j, start, stop;
   double currentTemp;
   int scanCounter;
   int deltaEnergy;
   MPI_Status status;
   struct timespec t1, t2;
   double startTime, endTime;

   start = startRow;
   stop = stopRow;

   if (myID == 0)
      start = startRow + 1;
   else if (myID == (numProcs - 1))
      stop = stopRow - 1;

   RandomizeSystem(startRow, stopRow, n);
```

```
currentTemp = STARTING_TEMP;

while (currentTemp >= ENDING_TEMP) {

   if (myID == 0)
      printf("Temp = %f\n", currentTemp);

   for (scanCounter = 0; scanCounter < LATTICE_SCANS; scanCounter++) {
      /* Send my disparity values to my neighbors and get values in. */

      if (myID == 0) {
         MPI_Send(disparityMap[stopRow], n, MPI_INT, myID + 1, myID,
            MPI_COMM_WORLD);
         MPI_Recv(disparityMap[stopRow+1], n, MPI_INT, myID + 1, myID + 1,
            MPI_COMM_WORLD, &status);
         }
      else if (myID == numProcs - 1) {
         MPI_Send(disparityMap[startRow], n, MPI_INT, myID - 1, myID,
            MPI_COMM_WORLD);
         MPI_Recv(disparityMap[startRow-1], n, MPI_INT, myID - 1, myID - 1,
            MPI_COMM_WORLD, &status);
         }
      else {
         MPI_Send(disparityMap[startRow], n, MPI_INT, myID - 1, myID,
            MPI_COMM_WORLD);
         MPI_Send(disparityMap[stopRow], n, MPI_INT, myID + 1, myID,
            MPI_COMM_WORLD);
         MPI_Recv(disparityMap[startRow-1], n, MPI_INT, myID - 1, myID - 1,
            MPI_COMM_WORLD, &status);
         MPI_Recv(disparityMap[stopRow+1], n, MPI_INT, myID + 1, myID + 1,
            MPI_COMM_WORLD, &status);
         }
   /* getclock(TIMEOFDAY, &t1);
   startTime = (t1.tv_sec * 100.0) + (t1.tv_nsec * 10e-7); */
      for (i = start; i <= stop; i++)
         for (j = 1; j < (n - D_MAX); j++) {
            newState = RandomNewState(disparityMap[i][j]);
            oldEnergy = Energy(i, j, disparityMap[i][j]);
            newEnergy = Energy(i, j, newState);
            deltaEnergy = newEnergy - oldEnergy;
            if (deltaEnergy < 0)
               tempDisparityMap[i][j] = newState;
            else if (RANDOM_PROBABILITY <
               exp((double)-deltaEnergy/currentTemp))
               tempDisparityMap[i][j] = newState;
            else
               tempDisparityMap[i][j] = disparityMap[i][j];
            }

      for (i = start; i <= stop; i++)
         for (j = 1; j < (n - D_MAX); j++)
            disparityMap[i][j] = tempDisparityMap[i][j];
   /*
   getclock(TIMEOFDAY, &t2);
   endTime = (t2.tv_sec * 100.0) + (t2.tv_nsec * 10e-7);
if (myID == 1)
printf("Elapsed time = %.2f seconds\n", (endTime - startTime) / 100.0);
   */

      }
   currentTemp -= (currentTemp * TEMP_REDUCTION);
   }

/* If I am not the boss, send my results back to the boss. */
```

```
   if (myID != 0)
      SendResultsToMaster(n, myID, startRow, stopRow);

   } /* end Anneal */


void main(int argc, char *argv[]) {
   int numProcs, myID, n;
   int startRow, stopRow;
   int chunkSize, destination;
   int i;
   double t1;

   MPI_Init(&argc, &argv);

   MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myID);

   n = atoi(argv[1]);

   /* Determine the start and stop rows. */

   chunkSize = (n / numProcs);
   startRow = (chunkSize * myID);
   stopRow = (startRow + chunkSize) - 1;
   if (myID == numProcs - 1)
      stopRow = max(stopRow, n-1);

   if (myID == 0) {
      /* Read in the left and right image files. */

      if (ReadLeftImage(n) && ReadRightImage(n)) {

         /* Send the image data to the different processors.  Processor 0
            gets the first part of the image. */

         for (i = chunkSize; i < n; i++) {
            destination = min(i / chunkSize, numProcs - 1);
            MPI_Send(leftImage[i], n, MPI_UNSIGNED_CHAR, destination, LEFT,
               MPI_COMM_WORLD);
            MPI_Send(rightImage[i], n, MPI_UNSIGNED_CHAR, destination,
               RIGHT, MPI_COMM_WORLD);
            }
         }

      t1 = MPI_Wtime();

      Anneal(n, myID, startRow, stopRow, numProcs);

      printf("Total time for row decomposition %d = %.2f seconds\n", n,
         MPI_Wtime() - t1);

      /* Collect results from the workers. */
      CollectResults(n, numProcs, chunkSize);

      WriteResultGrayScaleMap(n);

      }
   else {

      /* Gather information from the boss. */

      GetImageDataFromMaster(n, myID, startRow, stopRow);
      Anneal(n, myID, startRow, stopRow, numProcs);
```

```
    }
MPI_Finalize();
} /* end main */
```

INTENTIONALLY LEFT BLANK.

D MPI 1-D column decomposition.

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>

#define MAX_ROWS 1000
#define MAX_COLS 1000
#define D_MIN 0
#define D_MAX 6
#define LAMBDA 5
#define STARTING_TEMP 100.0
#define ENDING_TEMP 1.0
#define TEMP_REDUCTION 0.1
#define LATTICE_SCANS 10

/* Define various macros and functions to inline code. */

#define RANDOM_DISPARITY (rand() % D_MAX)
#define RANDOM_PROBABILITY (rand() % 32767 / 32767.0)
#define min(x,y) ((x) < (y)) ? (x) : (y)
#define max(x,y) ((x) > (y)) ? (x) : (y)

typedef unsigned char Pixel;
enum {LEFT = 0, RIGHT};

/* Global variables: */

Pixel leftImage[MAX_ROWS][MAX_COLS];
Pixel rightImage[MAX_ROWS][MAX_COLS];
int disparityMap[MAX_ROWS][MAX_COLS];
int tempDisparityMap[MAX_ROWS][MAX_COLS];

/* Function prototypes: */

/* Basic file I/O routines. */

static int ReadLeftImage(int n);
static int ReadRightImage(int n);
static void WriteResultGrayScaleMap(int n);

/* MPI communication routines. */

static void GetImageDataFromMaster(int n, int myID, int numProcs, int startCol, int stopCol);
static void SendResultsToMaster(int n, int myID, int startCol, int stopCol);
static void CollectResults(int n, int numProcs, int chunkSize);

/* Simulated annealing functions. */

static int RandomNewState(int oldState);
static void Anneal(int n, int myID, int startCol, int stopCol, int numProcs);

void main(int argc, char *argv[]);


/*****************************************************************************
   File I/O routines follow.
*****************************************************************************/

static int ReadLeftImage(int n) {
   FILE *theFile;
   char fileName[80];
   int i, j;
```

```
   /* Attempt to read the left image. */

   sprintf(fileName, "LeftDot%d.c", n);

   theFile = fopen(fileName, "r");

   if (theFile == NULL) {
      perror("ReadLeftImage: fopen");
      return(FALSE);
      }

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         fscanf(theFile, "%c", &(leftImage[i][j]));

   fclose(theFile);

   return(TRUE);

   } /* end ReadLeftImage */


static int ReadRightImage(int n) {
   FILE *theFile;
   char fileName[100];
   int i, j;

   /* Attempt to read the right image. */

   sprintf(fileName, "RightDot%d.c", n);

   theFile = fopen(fileName, "r");

   if (theFile == NULL) {
      perror("ReadRightImage: fopen");
      return(FALSE);
      }

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         fscanf(theFile, "%c", &(rightImage[i][j]));

   fclose(theFile);

   return(TRUE);

   } /* end ReadRightImage */


static void WriteResultGrayScaleMap(int n) {
   FILE *theFile;
   char fileName[80];
   int i, j;

   sprintf(fileName, "MPI_COLS_RESULTS_%d", n);

   theFile = fopen(fileName, "w");

   if (theFile == NULL) {
      perror("WriteResult: fopen");
      exit(1);
      }

   for (i = 0; i < n; i++)
```

```
      for (j = 0; j < n; j++)
         fprintf(theFile, "%c", (Pixel)(disparityMap[i][j] * (255 / D_MAX)));

   fclose(theFile);

   } /* end WriteResultGrayScaleMap */


/*************************************************************************
   MPI communication routines follow.
*************************************************************************/

static void CollectResults(int n, int numProcs, int chunkSize) {
   int i, source;
   MPI_Status status;
   MPI_Datatype column;

   MPI_Type_vector(n, 1, MAX_COLS, MPI_INT, &column);
   MPI_Type_commit(&column);

   for (i = chunkSize; i < n; i++) {
      source = min(i / chunkSize, numProcs - 1);
      MPI_Recv(&(disparityMap[0][i]), 1, column, source, source,
         MPI_COMM_WORLD, &status);
      }

   } /* end CollectResults */


static void GetImageDataFromMaster(int n, int myID, int numProcs, int startCol, int stopCol) {
   MPI_Status status;
   int i;
   MPI_Datatype column;

   MPI_Type_vector(n, 1, MAX_COLS, MPI_UNSIGNED_CHAR, &column);
   MPI_Type_commit(&column);

   for (i = startCol; i <= stopCol; i++) {
      MPI_Recv(&(leftImage[0][i]), 1, column, 0, LEFT, MPI_COMM_WORLD,
         &status);
      MPI_Recv(&(rightImage[0][i]), 1, column, 0, RIGHT, MPI_COMM_WORLD,
         &status);
      }

   /* We have to receive more of the left image since the disparity is
      by columns and we haven't received all the possible matched points. */

   if (myID != (numProcs - 1))
      for (i = stopCol + 1; i <= stopCol + D_MAX; i++)
         MPI_Recv(&(leftImage[0][i]), 1, column, 0, LEFT, MPI_COMM_WORLD,
            &status);

   } /* end GetImageDataFromMaster */


static void SendResultsToMaster(int n, int myID, int startCol, int stopCol) {
   int i;
   MPI_Datatype column;

   MPI_Type_vector(n, 1, MAX_COLS, MPI_INT, &column);
   MPI_Type_commit(&column);

   for (i = startCol; i <= stopCol; i++)
      MPI_Send(&(disparityMap[0][i]), 1, column, 0, myID, MPI_COMM_WORLD);
```

```
   } /* end SendResultsToMaster */


/*****************************************************************************
   Stereo matching routines follow.
*****************************************************************************/

static int RandomNewState(int oldState)
   {
   int randomNumber;

   if (RANDOM_DISPARITY > ((D_MAX / 2) - 1))
      randomNumber = -1;
   else
      randomNumber = 1;

   oldState = oldState + randomNumber;

   if (oldState < D_MIN)
      oldState = D_MIN;
   else if (oldState > D_MAX)
      oldState = D_MAX;

   return(oldState);

   } /* end RandomNewState */


static void RandomizeSystem(int startCol, int stopCol, int n) {
   int i, j;

   /* Assign a random disparity to each disparity grid point.  Edges get
      the value 0. */

   for (i = 0; i < n; i++)
      for (j = startCol; j <= stopCol; j++) {
         if ((i == 0) || (i == (n - 1)) || (j == 0) || (j >= (n - D_MAX)))
            disparityMap[i][j] = 0;
         else
            disparityMap[i][j] = RANDOM_DISPARITY;
         }

   } /* end RandomizeSystem */


static int Energy(int row, int col, int disparity) {
   int delta;
   int photometric;

   delta = abs(disparity - disparityMap[row-1][col-1]) +
           abs(disparity - disparityMap[row-1][col]) +
           abs(disparity - disparityMap[row-1][col+1]) +
           abs(disparity - disparityMap[row][col-1]) +
           abs(disparity - disparityMap[row][col+1]) +
           abs(disparity - disparityMap[row+1][col-1]) +
           abs(disparity - disparityMap[row+1][col]) +
           abs(disparity - disparityMap[row+1][col+1]);

   photometric = abs(leftImage[row][col+disparity] - rightImage[row][col]);

   return(photometric + (LAMBDA * delta));

   } /* end Energy */
```

```
static void Anneal(int n, int myID, int startCol, int stopCol, int numProcs) {
   int newState, newEnergy, oldEnergy, i, j, start, stop;
   double currentTemp;
   int scanCounter;
   int deltaEnergy;
   MPI_Status status;
   MPI_Datatype column;

   start = startCol;
   stop = stopCol;
   if (myID == 0)
      start = startCol + 1;
   else if (myID == (numProcs - 1))
      stop = (n - D_MAX) - 1;

   MPI_Type_vector(n, 1, MAX_COLS, MPI_INT, &column);
   MPI_Type_commit(&column);

   RandomizeSystem(startCol, stopCol, n);

   currentTemp = STARTING_TEMP;

   while (currentTemp >= ENDING_TEMP) {

      if (myID == 0)
         printf("Temp = %f\n", currentTemp);

      for (scanCounter = 0; scanCounter < LATTICE_SCANS; scanCounter++) {
         /* Send my disparity values to my neighbors and get values in. */

         if (myID == 0) {
            MPI_Send(&(disparityMap[0][stopCol]), 1, column, myID + 1, myID,
               MPI_COMM_WORLD);
            MPI_Recv(&(disparityMap[0][stopCol+1]), 1, column, myID + 1,
               myID + 1, MPI_COMM_WORLD, &status);
            }
         else if (myID == numProcs - 1) {
            MPI_Send(&(disparityMap[0][startCol]), 1, column, myID - 1, myID,
               MPI_COMM_WORLD);
            MPI_Recv(&(disparityMap[0][startCol-1]), 1, column, myID - 1,
               myID - 1, MPI_COMM_WORLD, &status);
            }
         else {
            MPI_Send(&(disparityMap[0][startCol]), 1, column, myID - 1, myID,
               MPI_COMM_WORLD);
            MPI_Send(&(disparityMap[0][stopCol]), 1, column, myID + 1, myID,
               MPI_COMM_WORLD);
            MPI_Recv(&(disparityMap[0][startCol-1]), 1, column, myID - 1,
               myID - 1, MPI_COMM_WORLD, &status);
            MPI_Recv(&(disparityMap[0][stopCol+1]), 1, column, myID + 1,
            myID + 1, MPI_COMM_WORLD, &status);
            }
         for (i = 1; i < (n - 1); i++)
            for (j = start; j <= stop; j++) {
               newState = RandomNewState(disparityMap[i][j]);
               oldEnergy = Energy(i, j, disparityMap[i][j]);
               newEnergy = Energy(i, j, newState);
               deltaEnergy = newEnergy - oldEnergy;
               if (deltaEnergy < 0)
                  tempDisparityMap[i][j] = newState;
               else if (RANDOM_PROBABILITY <
                  exp((double)-deltaEnergy/currentTemp))
                  tempDisparityMap[i][j] = newState;
               else
                  tempDisparityMap[i][j] = disparityMap[i][j];
```

```
            }

         for (i = 1; i < (n - 1); i++)
            for (j = start; j <= stop; j++)
               disparityMap[i][j] = tempDisparityMap[i][j];

         }
      currentTemp -= (currentTemp * TEMP_REDUCTION);
      }

   /* If I am not the boss, send my results back to the boss. */

   if (myID != 0)
      SendResultsToMaster(n, myID, startCol, stopCol);

   } /* end Anneal */


void main(int argc, char *argv[]) {
   int numProcs, myID, n;
   int startCol, stopCol;
   int chunkSize, destination;
   int i, j, oldDestination;
   double t1;
   MPI_Datatype column;

   MPI_Init(&argc, &argv);

   MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myID);

   n = atoi(argv[1]);

   /* Determine the start and stop columns. */

   chunkSize = (n / numProcs);
   startCol = (chunkSize * myID);
   stopCol = (startCol + chunkSize) - 1;
   if (myID == numProcs - 1)
      stopCol = max(stopCol, n-1);

   if (myID == 0) {
      /* Read in the left and right image files. */

      if (ReadLeftImage(n) && ReadRightImage(n)) {

         /* Send the image data to the different processors.  Processor 0
            gets the first part of the image. */

         MPI_Type_vector(n, 1, MAX_COLS, MPI_UNSIGNED_CHAR, &column);
         MPI_Type_commit(&column);

         oldDestination = 1;
         for (i = chunkSize; i < n; i++) {
            destination = min(i / chunkSize, numProcs - 1);
            if (destination != oldDestination) {
               for (j = i - 1; j <= (i - 1) + D_MAX; j++)
                  MPI_Send(&(leftImage[0][i]), 1, column, oldDestination,
                     LEFT, MPI_COMM_WORLD);
               oldDestination = destination;
               }
            MPI_Send(&(leftImage[0][i]), 1, column, destination, LEFT,
               MPI_COMM_WORLD);
            MPI_Send(&(rightImage[0][i]), 1, column, destination, RIGHT,
               MPI_COMM_WORLD);
```

```
            }
         }

      t1 = MPI_Wtime();

      Anneal(n, myID, startCol, stopCol, numProcs);

      printf("Total time for column decomposition %d = %.2f seconds\n", n,
         MPI_Wtime() - t1);

      /* Collect results from the workers. */
      CollectResults(n, numProcs, chunkSize);

      WriteResultGrayScaleMap(n);

      }
   else {

      /* Gather information from the boss. */

      GetImageDataFromMaster(n, myID, numProcs, startCol, stopCol);
      Anneal(n, myID, startCol, stopCol, numProcs);

      }

   MPI_Finalize();

   } /* end main */
```

INTENTIONALLY LEFT BLANK.

E FORTRAN and HPF code.

```
module timer

contains

  subroutine print_time
    character*8    :: date    ! ccyymmdd
    character*10   :: time    ! hhmmss.sss
    call date_and_time(date,time)
    print *, date(5:6)//'/'//date(7:8)//'/'//date(3:4)
    print *, time(1:2)//':'//time(3:4)//':'//time(5:10)
    return
  end subroutine print_time

  real function cputime()
    cputime = secnds(0.0)
    return
  end function cputime

end module timer

program rows
   use timer

   implicit none

   integer MAX_ROWS, MAX_COLS
   parameter (MAX_ROWS = 600, MAX_COLS = 600)
   integer D_MIN, D_MAX
   parameter (D_MIN = 0, D_MAX = 6)
   integer LAMBDA
   parameter (LAMBDA = 5)
   integer SCANS
   parameter (SCANS = 10)
   real STARTING_TEMP, ENDING_TEMP, TEMP_REDUCTION
   parameter (STARTING_TEMP = 100.0, ENDING_TEMP = 1.0)
   parameter (TEMP_REDUCTION = 0.1)
   double precision start, stop

   integer dMap(MAX_ROWS, MAX_COLS)
   integer, dimension(:,:), allocatable :: leftImage
   integer, dimension(:,:), allocatable :: rightImage
   integer, dimension(:,:), allocatable :: newDMap
   integer, dimension(:,:), allocatable :: oldEnergies
   integer, dimension(:,:), allocatable :: newEnergies
   real, dimension(:,:), allocatable :: randoms
!hpf$ distribute(block, *) :: dMap
!hpf$ align with dMap :: randoms
!hpf$ align with dMap :: newEnergies, oldEnergies, newDMap
!hpf$ align with dMap :: leftImage, rightImage
   real currentTemp
   integer i, j, n, scanCounter, k, l
   real dMaxReal
   interface
      pure integer function NewState(oldState, D_MIN, D_MAX)
         integer, intent(in) :: oldState, D_MIN, D_MAX
      end function NewState
   end interface
   interface
      pure real function rand()
      end function rand
   end interface
```

```
interface
   pure integer function irand()
   end function irand
end interface
interface
   pure integer function GetDisparity(current, proposed,newE,oldE, t)
   integer, intent(in) :: current, proposed, newE, oldE
   real, intent(in) :: t
   end function GetDisparity
end interface

allocate (leftImage(MAX_ROWS, MAX_COLS))
allocate (rightImage(MAX_ROWS, MAX_COLS))
allocate (newDMap(MAX_ROWS, MAX_COLS))
allocate (randoms(MAX_ROWS, MAX_COLS))
allocate (oldEnergies(MAX_ROWS, MAX_COLS))
allocate (newEnergies(MAX_ROWS, MAX_COLS))

n = 600
dMaxReal = D_MAX

call ReadLeftImage(n, leftImage)
call ReadRightImage(n, rightImage)

start = secnds(0.0)

! Initialize the disparity map

call random_number(randoms)

dMap(1,:) = 0
dMap(n,:) = 0
dMap(:,1) = 0
dMap(:,(n-D_MAX)+1:n) = 0
forall (i=2:n-1, j=2:n-D_MAX)
   dMap(i,j) = mod(randoms(i,j) * 10000, dMaxReal)
end forall

currentTemp = STARTING_TEMP

! Start the annealing process

do while (currentTemp >= ENDING_TEMP)
   print *, 'Temp = ', currentTemp
   do scanCounter = 1,SCANS

   forall (i=2:n-1, j=2:n-D_MAX)
   newDMap(i,j) = NewState(dMap(i,j), D_MIN, D_MAX)

   oldEnergies(i,j) = ((abs(dMap(i-1,j-1) - dMap(i,j)) +          &
  &                    abs(dMap(i-1,j) - dMap(i,j)) +             &
  &                    abs(dMap(i-1,j+1) - dMap(i,j)) +           &
  &                    abs(dMap(i,j-1) - dMap(i,j)) +             &
  &                    abs(dMap(i,j+1) - dMap(i,j)) +             &
  &                    abs(dMap(i+1,j-1) - dMap(i,j)) +           &
  &                    abs(dMap(i+1,j) - dMap(i,j)) +             &
  &                    abs(dmap(i+1,j+1) - dMap(i,j))) *          &
  &                    LAMBDA) + abs(leftImage(i,j+dMap(i,j)) -   &
  &                    rightImage(i,j))

   newEnergies(i,j) = ((abs(dMap(i-1,j-1) - newDMap(i,j)) +       &
  &                    abs(dMap(i-1,j) - newDMap(i,j)) +          &
  &                    abs(dMap(i-1,j+1) - newDMap(i,j)) +        &
  &                    abs(dMap(i,j-1) - newDMap(i,j)) +          &
  &                    abs(dMap(i,j+1) - newDMap(i,j)) +          &
```

```
&                   abs(dMap(i+1,j-1) - newDMap(i,j)) +          &
&                   abs(dMap(i+1,j) - newDMap(i,j)) +            &
&                   abs(dmap(i+1,j+1) - newDMap(i,j))) *         &
&                   LAMBDA) + abs(leftImage(i,j+newDMap(i,j)) -  &
&                   rightImage(i,j))

   randoms(i,j) = rand()

   end forall

   forall (i=2:n-1, j=2:n-D_MAX, (newEnergies(i,j) -              &
  &  oldEnergies(i,j) < 0) .or. (randoms(i, j) < exp(             &
  & (newEnergies(i,j) - oldEnergies(i,j)) / -currentTemp)))
      dMap(i,j) = newDMap(i,j)
   end forall

   enddo

   currentTemp = currentTemp - (currentTemp * TEMP_REDUCTION)

enddo

stop = secnds(0.0)

print *, 'Total time parallel = ', stop - start

call WriteResults(n, dMap)

end


    subroutine ReadLeftImage(n, leftImage)
    implicit none
    integer n
    integer leftImage(n, n)
    integer i, j

    open (unit=10, file='LeftDot600.f', status='old')
    read (10,*) ((leftImage(i,j), j=1,n),i=1,n)
    close (10)

    return
    end


    subroutine ReadRightImage(n, rightImage)
    implicit none
    integer n
    integer rightImage(n, n)
    integer i, j

    open (unit=10, file='RightDot600.f', status='old')
    read (10,*) ((rightImage(i,j), j=1,n),i=1,n)
    close (10)

    return
    end


    subroutine WriteResults(n, disparityMap)
    implicit none
    integer n
    integer disparityMap(n, n)
    integer i, j
```

```
      open (unit=10, file='HPF_ROW_RESULTS', status='new')
      write (10,*) (((disparityMap(i,j) * (255 / 6)), j=1,n),i=1,n)
      close (10)

      end subroutine WriteResults


pure integer function NewState(oldState, D_MIN, D_MAX)
   implicit none
   intent(in) :: oldState, D_MIN, D_MAX
   integer oldState
   integer D_MIN, D_MAX
   integer randomNumber
   real x
   interface
      pure integer function irand()
      end function irand
   end interface


  if (mod(irand(), D_MAX) > ((D_MAX / 2) - 1)) then
     randomNumber = -1
  else
     randomNumber = 1
  end if

      NewState = oldState + randomNumber

      if (NewState < D_MIN) then
         NewState = D_MIN
      else if (NewState > D_MAX) then
         NewState = D_MAX
      end if

      return
      end function


pure integer function GetDisparity(current, proposed, newE, oldE, t)
   implicit none
   intent(in) :: current, proposed, newE, oldE
   intent(in) :: t
   integer current, proposed, newE, oldE
   real t
   interface
      pure real function rand()
      end function rand
   end interface

   if ((newE - oldE) < 0) then
      GetDisparity = proposed
   else if (rand() < exp((-(newE-oldE)) / t)) then
      GetDisparity = proposed
   else
      GetDisparity = current
   end if

   return
   end function
```

NO. OF COPIES ORGANIZATION

2 DEFENSE TECHNICAL INFORMATION CENTER
DTIC DDA
8725 JOHN J KINGMAN RD
STE 0944
FT BELVOIR VA 22060-6218

1 HQDA
DAMO FDQ
DENNIS SCHMIDT
400 ARMY PENTAGON
WASHINGTON DC 20310-0460

1 DPTY ASSIST SCY FOR R&T
SARD TT F MILTON
RM 3EA79 THE PENTAGON
WASHINGTON DC 20310-0103

1 OSD
OUSD(A&T)/ODDDR&E(R)
J LUPO
THE PENTAGON
WASHINGTON DC 20301-7100

1 CECOM
SP & TRRSTRL COMMCTN DIV
AMSEL RD ST MC M
H SOICHER
FT MONMOUTH NJ 07703-5203

1 PRIN DPTY FOR TCHNLGY HQ
US ARMY MATCOM
AMCDCG T
M FISETTE
5001 EISENHOWER AVE
ALEXANDRIA VA 22333-0001

1 DPTY CG FOR RDE HQ
US ARMY MATCOM
AMCRD
BG BEAUCHAMP
5001 EISENHOWER AVE
ALEXANDRIA VA 22333-0001

1 INST FOR ADVNCD TCHNLGY
THE UNIV OF TEXAS AT AUSTIN
PO BOX 202797
AUSTIN TX 78720-2797

NO. OF COPIES ORGANIZATION

1 GPS JOINT PROG OFC DIR
COL J CLAY
2435 VELA WAY STE 1613
LOS ANGELES AFB CA 90245-5500

1 ELECTRONIC SYS DIV DIR
CECOM RDEC
J NIEMELA
FT MONMOUTH NJ 07703

3 DARPA
L STOTTS
J PENNELLA
B KASPAR
3701 N FAIRFAX DR
ARLINGTON VA 22203-1714

1 US MILITARY ACADEMY
MATH SCI CTR OF EXCELLENCE
DEPT OF MATHEMATICAL SCI
MDN A MAJ DON ENGEN
THAYER HALL
WEST POINT NY 10996-1786

1 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CS AL TP
2800 POWDER MILL RD
ADELPHI MD 20783-1145

1 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CS AL TA
2800 POWDER MILL RD
ADELPHI MD 20783-1145

3 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CI LL
2800 POWDER MILL RD
ADELPHI MD 20783-1145

ABERDEEN PROVING GROUND

4 DIR USARL
AMSRL CI LP (305)

INTENTIONALLY LEFT BLANK.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1998	Final, October 1997 - December 1997

4. TITLE AND SUBTITLE

MPI and HPF Performance in a DEC Alpha Cluster

5. FUNDING NUMBERS

78M841

6. AUTHOR(S)

Dale Shires

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

U.S. Army Research Laboratory
ATTN: AMSRL-WM-MD
Aberdeen Proving Ground, MD 21005-5066

8. PERFORMING ORGANIZATION REPORT NUMBER

ARL-TR-1668

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

10.SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

There are several different types of parallel computer architectures in use today. Some of these are large machines that house hundreds of processors with low- to mid-range computing capabilities. A different type of parallel computer architecture becoming increasingly popular is that of the cluster. Clusters are basically networked workstations: each containing 1 to ~30 processors with mid- to high-range computing capabilities. While arguments can be made for both paradigms, clusters seem to be gaining in popularity. They provide fast computation through multiplicity and fast processor throughput. Furthermore, code reuse on different cluster environments is now possible with the adoption of standard interprocessor communication tools like the message-passing interface (MPI) and high-performance FORTRAN (HPF). This report compares and contrasts the performance of MPI and HPF on a currently available computer cluster.

14. SUBJECT TERMS

clusters, parallel computing, high-performance FORTRAN, message-passing interface

15. NUMBER OF PAGES

50

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

INTENTIONALLY LEFT BLANK.

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number/Author ARL-TR-1668 (Shires) Date of Report May 1998

2. Date Report Received _____

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) _____

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.) _____

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. _____

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) _____

CURRENT ADDRESS

_____
Organization

_____
Name E-mail Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

OLD ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)

(DO NOT STAPLE)